CPD - Performance Evaluation of a single core

2022/2023

Neste projeto, iremos realizar um estudo sobre o efeito da performance do processador na memória quando acedemos grandes quantidades de dados.

Índice

Descrição do problema e explicação dos algoritmos

O objetivo deste projeto é comparar e analisar o desempenho de um cpu em diferentes situações apresentadas. Como é do nosso conhecimento, esse desempenho depende de diversos fatores tais como a gestão de memória num determinado programa, a utilização de diferentes algoritmos, a linguagem de programação utilizada e até das próprias características físicas do processador.

Como tal, iremos apresentar neste relatório algumas abordagens em relação à multiplicação de matrizes e identificar as principais diferenças entre cada uma delas. Para a realização destes testes utilizamos a API PAPI assim como C++ e Java como linguagens de programação.

Multiplicação de matrizes pelo método algébrico

O primeiro exercício deste trabalho envolvia a multiplicação de matrizes pelo método mais utilizado, ou seja, o produto das matrizes A = (aij) (m x p) e B = (bij) (p x n) seria a matriz C = (cij) (m x n), em que cada elemento cij é calculado com a soma dos produtos dos elementos da linha i de A com os elementos da coluna j de B.

$$\begin{bmatrix} 1 & 3 \\ 2 & 5 \end{bmatrix} \times \begin{bmatrix} 2 & 2 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1\times 2+3\times 0 & \\ & \end{bmatrix}$$

```
for(i=0; i<m_ar; i++){
  for(j=0; j<m_br; j++){
    temp = 0;
    for(k=0; k<m_ar; k++){
      temp += pha[i*m_ar+k] * phb[k*m_br+j];
    }
    phc[i*m_ar+j]=temp;
  }
}
```

Multiplicação por linhas

Se no primeiro exercício multiplicamos os elementos da linha i de A pelos elementos da coluna j de B, nesta segunda alínea iremos multiplicar os elementos da linha i de A pelos elementos da linha i de B, verificando se existe alguma diferença de desempenho por parte do cpu.

```
for(i=0; i<m_ar; i++)
  for( k=0; k<m_ar; k++)
    for( j=0; j<m_br; j++)
      phc[i*m_ar+j] += pha[i*m_ar+k] * phb[k*m_br+j];
```

Multiplicação em bloco

Agora, neste terceiro exercício, temos como objetivo dividir as nossas matrizes A e B em várias submatrizes e de seguida calcular a matriz resultado. Para isso, utilizaremos o primeiro método, ou seja, o método algébrico. Iremos, portanto, analisar se o facto de dividir a matriz em partes mais pequenas contribui para diferenças no número de operações realizadas pelo processador.

```
for (int ii = 0; ii < m_ar; ii += bkSize)
  for (int jj = 0; jj < m_br; jj += bkSize)
    for (int kk = 0; kk < m_ar; kk += bkSize)
      // Perform multiplication on blocks of size bkSize
      for (int i = 0; i < bkSize; i++)
        for (int k = 0; k < bkSize; k++)
          for (int j = 0; j < bkSize; j++)
            c[(i+ii) * m_ar + (j+jj)] += a[(i+ii) * m_ar + (k+kk)] * b[(k+kk) *
m_br + (j+jj)];
```

Métricas de Desempenho

Tal como nos foi proposto, para medir o desempenho do processador no cálculo de multiplicação de matrizes utilizamos duas linguagens de programação diferentes: C++ e Java. Com isto, conseguimos medir as diferenças no tempo que cada programa demorou a correr.

Para além disso utilizamos o PAPI (Performance API) de forma a coletar valores úteis como o número de cache misses nos níveis L1 e L2 da cache.

Na avaliação de resultados tivemos também em conta a dimensão das matrizes pelo que foi um dos aspetos propostos para este trabalho.

Resultados e Análise

1. **Multiplicação de matrizes pelo método algébrico**

C++ Performance:

Size	Time (s)	Level 1 DCM	Level 2 DCM
600	0,182	244776051	37737956
1000	0,977	1228428924	214386260
1400	3,057	3508798196	530845856
1800	16,808	9092784330	3150759350
2200	37,375	17631135986	18512819143
2600	67,284	30907455725	46183298297
3000	113,543	50296860285	94674304564

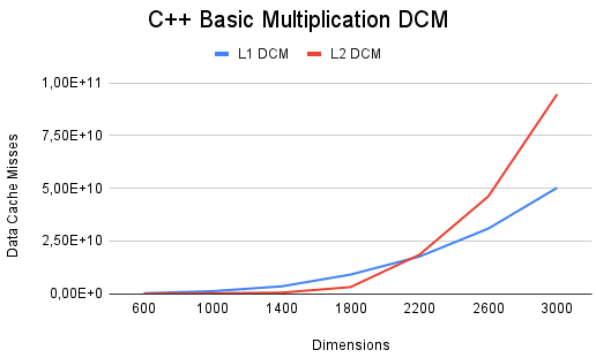


Figura 1: C++ Basic Multiplication DCM

Através da análise dos dados da tabela, é possível concluir que o tamanho da matriz e o tempo de execução são diretamente proporcionais. Em relação aos *data cache misses* podemos seguir a mesma conclusão, sabendo que quanto maior a matriz mais memória é acessada.

Java Performance:

Size	Time (s)	1800	18,959
600	0,443	2200	39,835
1000	1,533	2600	70,101
1400	4,974	3000	114,679

Tal como na prévia análise, sabemos que com o o aumento do tamanho da matriz é possível verificar um acréscimo de tempo na sua execução.

1. **Multiplicação em Linha**

C++ Performance:

Size	Time (s)	Level 1 DCM	Level 2 DCM
600	0,091	27102899	56652910
1000	0,43	125786944	261062955
1400	1,514	346297957	700121534
1800	3,183	745277442	1425674947
2200	6,161	2072120207	2515419153
2600	10,27	4411931302	4113837413
3000	15,753	6779037601	6332366591

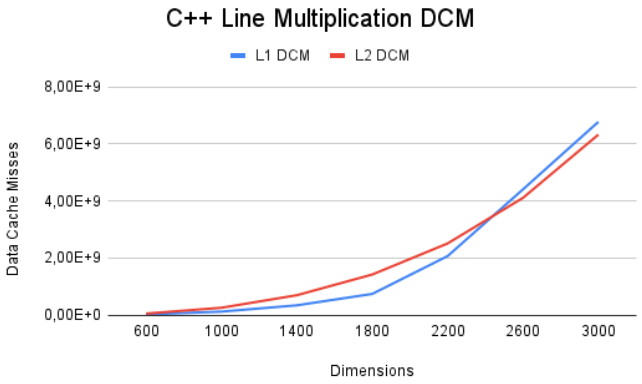


Figura 2: C++ Line Multiplication DCM

Size	Time (s)	Level 1 DCM	Level 2 DCM

4096	40,279	17547752512	15707309470
6144	135,786	59147235476	52734959866
8192	322,72	140073542249	125068329036
10240	628,837	273793089062	250892242727

Em comparação com a multiplicação através do método algébrico, há uma melhoria nas métricas de desempenho, devido ao algoritmo tirar vantagem dos valores já alocados em cache. A utilização de um diferente algoritmo permite a obtenção de melhores tempos e de menores *data cache misses*.

Java Performance:

Size	Time (s)	1800	4,43
600	0,362	2200	10,978
1000	0,581	2600	18,211
1400	1,824	3000	28,094

Em Java, apesar de o tempo de execução ser menor com a utilização de outro algoritmo, ainda existe um acréscimo ao de tempo na sua execução em comparação com C++.

C++ vs Java:

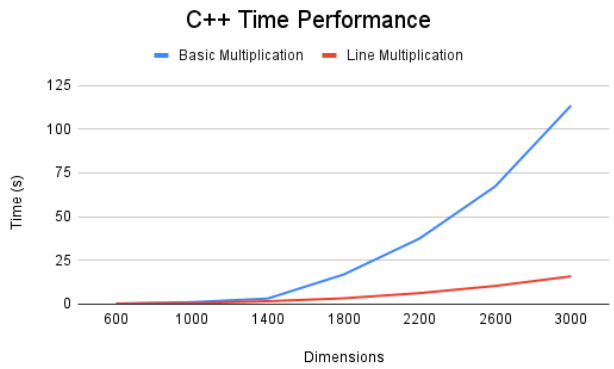


Figura 2: C++ Time Performance

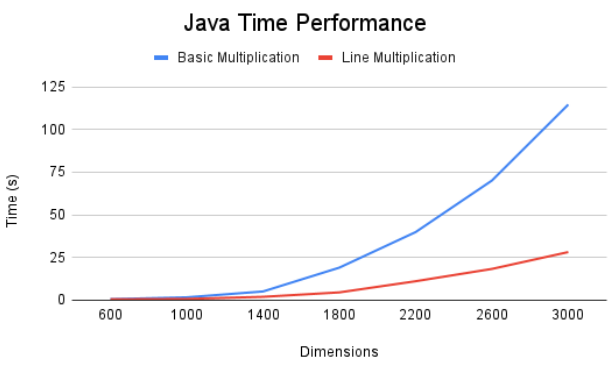


Figura 3: Java Time Performance

A partir dos gráficos acima e dos resultados obtidos, é possível verificar que para ambas as linguagens os tempos são comparáveis na execução dos algoritmos, no entanto, em C++ é ligeiramente mais rápido.

1. **Multiplicação por Blocos**

C++ Performance:

Block Size = 128

Size	Time (s)	Level 1 DCM	Level 2 DCM
4096	38,866	9816072352	33532269676

6144	131,067	9131318958	112528948104
8192	281,226	8765175603	262099357363
10240	618,003	33119770223	508501463014

Block Size = 256

Size	Time (s)	Level 1 DCM	Level 2 DCM
4096	34,87	30810947444	23616455632
6144	118,091	29605837694	76446923679
8192	402,54	78525145239	160980671603
10240	564,167	73089269641	352351742155

Block Size = 512

Size	Time (s)	Level 1 DCM	Level 2 DCM
4096	40,795	7022062687	19395802889
6144	107,541	153312269817	66233153737
8192	341,662	142601487187	138084006642
10240	514,313	136893056661	307158357141

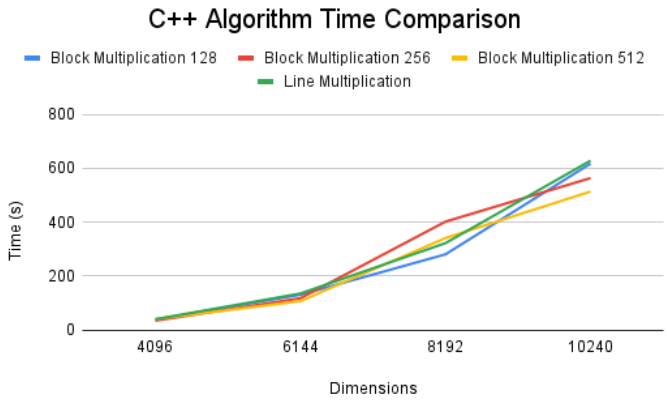
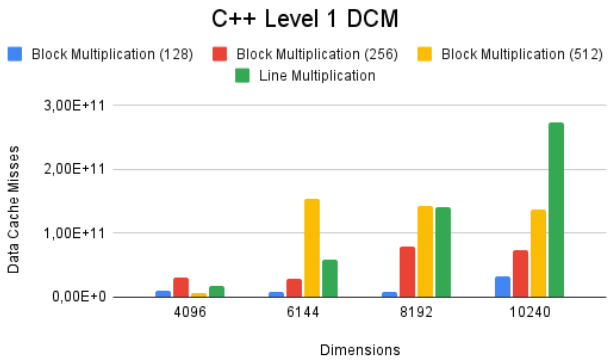


Figura 4: C++ Algorithm Time Comparison



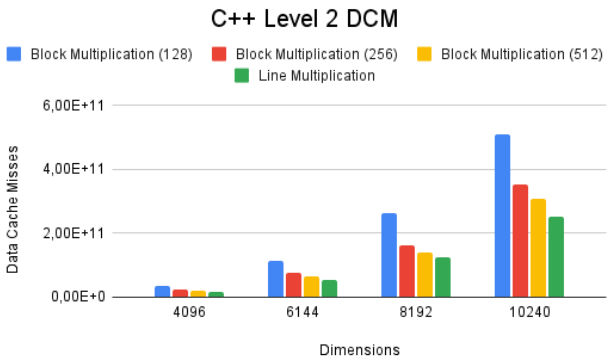


Figura 5: C++ Level 1 DCM

Figura 6: C++ Level 2 DCM

Este tipo de abordagem pretendia tirar partido do menor número de chamadas à memória. Isto acontecia devido ao facto de dividirmos a matriz em blocos mais pequenos, o que por sua vez subdivide o problema. Por fim, a multiplicação por blocos resulta em números de cache misses mais baixos.

Durante os testes neste tipo de multiplicação de matrizes, usamos três tamanhos de blocos para as subdividir: 128, 256 e 512. O que pudemos observar foi que à medida que aumentamos o tamanho desses blocos o desempenho iria aumentar na maioria das vezes.

Conclusão

De certa forma este trabalho contribuiu para uma melhor perceção acerca do desempenho de um processador e como vários fatores o podem influenciar, desde diferentes algoritmos, alocação mais eficiente de memória (o que nos leva também à linguagem de programação utilizada) e por fim, aos diferentes métodos de gerência de processos, neste caso relativo à multiplicação de matrizes.

Estudantes

- Afonso Abreu 202008552
- Raquel Carneiro 202005330